Тема: Реалізація ефектів руху (Action)

Мета: Засвоїти основні команди для реалізації руху спрайтів. Освоїти особливості використання анімації зміни зовнішнього вигляду об'єктів.

Зміст:

1. Найпростіші команди руху
2. Обертання (Rotate)
3. Масштабування (Scale)
4. Зникання та поява
5. Зміна відтінку (Tint)
6. Стрибки (jump)
7. Анімація спрайтів.
8. Прискорення та затримка руху (Easing)
9. Послідовності
10. Clone
11. Реверс

Найпростіші команди руху

Без руху та анімацій персонажів важко уявити собі будь-яку гру. В cocos2d-x існують спеціальні методи, які дозволяють здійснити ефекти переміщення, стрибка, поворотів, кручень, динамічної зміни масштабу. Всі ці методи роблять вузол зміною своїх властивостей. Об'єкти дії дозволяють перетворювати властивості вузла(положення, масштаб, нахил, колір) за вказаний період часу. Будь-який об'єкт з базового класу може використовувати об'єкти дії. Як приклад, можна перемістити Sprite з однієї позиції в іншу і робити це протягом певного проміжку часу.

Найпоширенішими методами є **MoveTo і MoveBy.**

Таблиця 1. Команди руху

`auto moveBy = MoveBy::create(t, Vec2(x, y));`	Створення об'єкта класу MoveBy , який означає зміну положення координат на вказані значення **x, y** , за час **t сек.**
`auto moveTo = MoveTo::create(t, Vec2(x, y));`	Створення об'єкта класу MoveTo , який означає зміну положення до вказаних значення **x, y** , за час **t сек**

Приклад використання

```
// рух спрайту до координат 50,10 за 2 секунди.
    auto moveTo = MoveTo::create(2, Vec2(50, 10));
    mySprite1->runAction(moveTo);
```

```
        // рух спрайту на 20 одиниць по осі oX за 2 секунди
        auto moveBy = MoveBy::create(2, Vec2(20, 0));
        mySprite2->runAction(moveBy);
```

Різниці між **moveTo** та **moveBy.**

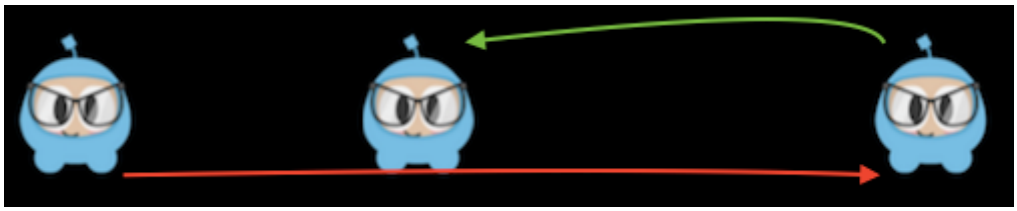
Обидві команди задають зміну положення спрайта. Причому ця зміна не буде миттєвою, а за вказаний період часу в секундах. І зміна положення буде розрахована рівномірно в залежності від величини зміни і часу зміни положення.

Команда MoveTo здійснює зміну положення спрайту поміщаючи його в позицію, яка вказана в аргументах функції Vec2 (). Функція MoveBy змінює положення спрайту на вказані аргументи функції Vec2 ().

В прикладі вище, якщо спрайти мав положення в координатах (100, 100), то після виконання команди MoveTo його положення буде в координатах (50, 10), а після виконання команди MoveBy його положення буде в координатах (70, 10)

Нище наведено, ще один приклад виконання дії команд переміщення:

```
auto mySprite = Sprite::create("mysprite.png");
        mySprite->setPosition(Vec2(200, 256));


        auto moveBy = MoveBy::create(2, Vec2(500, mySprite->getPositionY()));


        auto moveTo = MoveTo::create(2, Vec2(300, mySprite->getPositionY()));

        // Створення об'єкту паузи, який також е елементом руху при виконання дій
        auto delay = DelayTime::create(1);

        // створення об'єкту послідовність, який об'єднює та запускає всі встановлені
дії
auto seq = Sequence::create(moveBy, delay, moveTo, nullptr);

        mySprite->runAction(seq);
```



мал. 1

Обертання (Rotate)

Ми вже зустрічались з командою для повороту вузла:

```
mySprite->setRotation(40);
```

Проте це статична команда. Існує команда, яка задаватиме зміну положення спрайту, а саме його повороту навколо точки прив'язки на заданий кут у вигляді послідовного обертання за заданий час. Формат команди:

```
auto rotateTo = RotateTo::create(t, g);
```

t- час зміни, g – кут зміни.

Слід зауважити, що сам ефект поворот, залежить від точки привязки спрайту. Поворот буде здійснюватись саме навколо точки привязки. Нище наведено два майже однакові коди, працюватимуть вони по різному.

```
auto mySprite = Sprite::create("3.png");
mySprite->setAnchorPoint(Vec2(0.5,
0.5));
Size size = Director::getInstance()-
>getWinSize();
mySprite->setPosition(Vec2(size.width /
2, size.height / 2));

auto rotateTo = RotateTo::create(3.0f,
90.0f);
mySprite->runAction(rotateTo);
```

```
auto mySprite = Sprite::create("3.png");
mySprite->setAnchorPoint(Vec2(0, 0));
Size size = Director::getInstance()-
>getWinSize();
mySprite->setPosition(Vec2(size.width /
2, size.height / 2));

auto rotateTo = RotateTo::create(3.0f,
90.0f);
mySprite->runAction(rotateTo);
```

Масштабування (Scale)

Аналогічно як з поворотом, з можливістю зміни маштабу ми вже зустрічались, використовуючи наступний код:

```
mySprite->setScale(2.0);
```

Проте це статична команда. Існує команда, яка задаватиме зміну маштабу спрайту, за заданий час. Існують кілька різновидів команди. Формати команд:

```
auto scaleBy = ScaleBy::create(t, g);
```

t- час зміни, g – коефіцієнт масштабу.

```
auto scaleBy = ScaleBy::create(t, x, y);
```

t- час зміни, x – зміна по ширині, y – зміна по висоті

```
auto scaleBy = ScaleTo::create(t, g);
```

t- час зміни, g – коефіцієнт масштабу.

```
auto scaleBy = ScaleTo::create(t, x, y);
```

t- час зміни, x – зміна по ширині, y – зміна по висоті

Різниця між формами **To** та **By** такаж як і в командах **moveTo** та **moveBy.**

Для порівняння методів між собою запустіть на виконання нищенаведений код.

```
auto scaleBy = ScaleBy::create(2.0f, 3.0f);
mySprite->runAction(scaleBy);

auto scaleBy1 = ScaleBy::create(2.0f, 3.0f, 3.0f);
mySprite->runAction(scaleBy1);

auto scaleTo = ScaleTo::create(2.0f, 3.0f);
```

```
mySprite->runAction(scaleTo);

auto scaleTo1 = ScaleTo::create(2.0f, 3.0f, 3.0f);
mySprite->runAction(scaleTo1);
```

Всі ці дії запускатимуться одночасно, тому доцільніше використати послідовність, щоб зміни відбувались одна за одною.

```
auto delay = DelayTime::create(1);
auto scaleBy = ScaleBy::create(2.0f, 3.0f);

auto scaleBy1 = ScaleBy::create(2.0f, 1.0f, 3.0f);

auto scaleTo = ScaleTo::create(2.0f, 3.0f);

auto scaleTo1 = ScaleTo::create(2.0f, 3.0f, 3.0f);

auto seq = Sequence::create(delay->clone(), scaleBy, delay,scaleBy1, delay-
>clone(), scaleTo, delay->clone(), scaleTo1, nullptr);

mySprite->runAction(seq);
```

Зникання та поява

Серед анімаційних ефектів в cocos2d-x з методи, які дозволяють приховати об'єкти з екрану, а потім їх знов відобразити. Ця властивість носить назву Fade, а методи FadeIn і FadeOut.

Ступінь непрозорості має 255 рівнів градації, відповідно 0 видимий, 255невидимий.

Формат запису команд:

```
auto fadeIn = FadeOut::create(t);
auto fadeIn = FadeIn::create(t);
```

t – час зникання чи появи (час за який вузол рівномірно пройде всі 255 рівнів градації).

Приклад:

```
// Зникання об'єкту за 3 секунди
auto fadeOut = FadeOut::create(3.0f);
mySprite->runAction(fadeOut);
```

мал. 2

Наступний код продемонструє зникання вузла під час руху:

```
auto fadeOut = FadeOut::create(5.0f);
mySprite->runAction(fadeOut);

auto moveTo = MoveTo::create(5f, Vec2(400, 300));
mySprite->runAction(moveTo);
```

Зміна відтінку (Tint)

Відтінок вузла, який реалізує протокол NodeRGB, дозволяє змінити забарвлення вузла від поточного відтінку до спеціального кольору. Для цього використовуються дві команди TintTo та TintBy

Формат запису команд:

```
auto action = TintBy::create(t, redValue, greenValue, blueValue);
playerSprite->runAction(action);

auto action = TintTo::create(t, redValue, greenValue, blueValue);
playerSprite->runAction(action);
```

Приклад використання:

```
auto tintTo = TintTo::create(5.0f, 255.0f, 0.0f, 255.0f);
mySprite->runAction(tintTo);
```

мал. 3

Стрибки (jump)

Дія стрибка, як випливає з назви, дозволяє вузлу стрибати по сцені протягом певного періоду часу, який можна задати. Існує два типи стрибкових дій:

• **JumpBy**: Ця команда вказує параметри стрибка вузла на вказані параметри, відносно його поточної позиції. Наприклад: вузол знаходиться в (400, 500), а дія JumpBy застосовується з параметрами (60, 70) з тривалістю 2 секунди; стрибаючи п'ять разів, при кожному стрибку, що досягає висоти 45 пікселів, вузол буде переходити до (460, 570) з початкового положення (400, 500) протягом періоду 2 секунд лінійним чином.

Синтаксис команди:

```
auto action = JumpBy::create(duration, Vec2(xPosition, yPosition),
heightOfEachJump, numberOfJumps);
```

```
playerSprite->runAction(action);
```

`duration` – загальний час на стрибки;
`xPosition` – позиція на яку має змінитись положення по oX
`yPosition` - позиція на яку має змінитись положення по oY
`heightOfEachJump` – висота стрибка
`numberOfJumps` – кількість стрибків

Час одного стрибка можна визначити як частку duration/numberOfJumps

Приклад коду:

```
auto action = JumpBy::create(5.0, Vec2(0, 0), 100, 5);
mySprite->runAction(action);
```

JumpTo: Ця дія змушує вказаний вузол перейти до заданої позиції з його поточної позиції. Наприклад, вузол знаходиться в (400, 500), а дія JumpTo застосовується з (60, 70) з тривалістю 2 секунди; стрибаючи 5 разів, при кожному стрибку, що досягає висоти 45 пікселів, вузол підскочить до (60, 70) з початкового положення (400, 500) протягом періоду 2 секунд у лінійному русі.
Синтаксис команди:

```
auto action = JumpTo::create(duration, Vec2(xPosition, yPosition),
heightOfEachJump, numberOfJumps);
playerSprite->runAction(action);
```

`duration` – загальний час на стрибки;
`xPosition` – позиція до якої має перейти вузол по oX
`yPosition` - позиція до якої має перейти вузол по oY
`heightOfEachJump` – висота стрибка
`numberOfJumps` – кіоткість стрибків

Час одного стрибка можна визначити як частку duration/numberOfJumps

Приклад коду:

```
auto action = JumpTo::create(5.0, Vec2(0, 0), 100, 5);
mySprite->runAction(action);
```

Анімація спрайтів.

З Animate можна зробити просту анімацію flipbook з об'єктами Sprite. Це просто замінює кадр відображення за заданими інтервалами протягом тривалості анімації. Давайте розглянемо цей приклад:

Коли персонажі в грі починають рухатися, гра вийде живим. Є багато можливостей створити анімаційні ефекти. З Animate можна зробити просту анімацію flipbook з об'єктами Sprite. Це просто замінює кадр відображення за заданими інтервалами протягом тривалості анімації.

Важливим елементом є формування масиву зображень, які будуть міняти одне одне з певною затримкою.

Найпоширеніший спосіб створити анімацію з серії послідовних зображень.

Для створення анімації, використовується клас **Animation** та клас **Animate**.

```
auto animation = Animation::create();
```

Вищенаведений код створює об'єкт класу Animation

Подальшими діями потрібно додати до об'єкту спрайти, це виконує метод **addSpriteFrameWithFile**, наступний код ілюструє додавання восьми спрайтів:

```
animation->addSpriteFrameWithFile("run/RUN_000.png");
animation->addSpriteFrameWithFile("run/RUN_001.png");
animation->addSpriteFrameWithFile("run/RUN_002.png");
animation->addSpriteFrameWithFile("run/RUN_003.png");
animation->addSpriteFrameWithFile("run/RUN_004.png");
animation->addSpriteFrameWithFile("run/RUN_005.png");
animation->addSpriteFrameWithFile("run/RUN_006.png");
```

Потім слід задати час затримки зміни кадрів:

```
animation->setDelayPerUnit(1.0f);
```

та, при потребі, кількість повторень:

```
animation->setLoops(10);
```

Якщо потрібно щоб анімація була постійна, тоді в методі **setLoops** потрібно встановити аргумент рівний -1:

```
animation->setLoops(-1);
```

Підкінець залишається створити об'єкт **Animate** на основі об'єкту **animation**, і нарешті запустити метод **runAction**, передавши в екземпляр **Animate**.

```
auto action = Animate::create(animation);
sprite->runAction(action);
```

Для вічного зациклення анімації, щоб не використовувати метод **animation->setLoops(-1);** можна використати функцію **RepeatForever**.

Наприклад

```
Animate* animate = Animate::create(animation);
mySprite->runAction(RepeatForever::create(animate));
```

В підсумку, для створення анімації руху потрібно скомпілювати наступний код:

```
auto mySprite = Sprite::create("run/RUN_000.png");
mySprite->setAnchorPoint(Vec2(0, 0));
mySprite->setPosition(Vec2(50, 300));
auto animation = Animation::create();
animation->addSpriteFrameWithFile("run/RUN_000.png");
animation->addSpriteFrameWithFile("run/RUN_001.png");
animation->addSpriteFrameWithFile("run/RUN_002.png");
animation->addSpriteFrameWithFile("run/RUN_003.png");
animation->addSpriteFrameWithFile("run/RUN_004.png");
animation->addSpriteFrameWithFile("run/RUN_005.png");
animation->addSpriteFrameWithFile("run/RUN_006.png");
animation->setDelayPerUnit(1.0f);
```

```
Animate* animate = Animate::create(animation);
mySprite->runAction(RepeatForever::create(animate));
scene->addChild(mySprite);
```

Для заповнення **animation** зображеннями-спрайтами в вищеописаному коді не можна використовувати частинки спрайтів, за допомогою **Rect**. Це можна зробити використовуючи **SpriteFrame**:

```
animFrames.pushBack(SpriteFrame::create("Blue_Front1.png", Rect(0, 0, 65, 81)));
```

в даному прикладі animFrames це об'єкт класу вектор (Vector).

Поелементне добавлення спрайтів можна замінити циклом:

```
auto animation = Animation::create();
    for (int i = 1; i <= 6; i++) { // від run_001.png до run_006.png
        std::string name = StringUtils::format("run/run_%002d.png", i);
        animation->addSpriteFrameWithFile(name.c_str());
    }
```

Клас StringUtils визначає певні слова, пов'язані з обробкою рядка.. StringUtils :: toString метод може генерувати std :: string значення з різних значень.

Поспрайтне добавлення зображень є громіздким і незручним. У таких випадках використовують клас-вектор куди прописують циклом всі потрібні файли, і мотім цей вектор записують в об'єкт **animation**

```
    Vector<SpriteFrame*> animFrames;
    animFrames.reserve(12);
    animFrames.pushBack(SpriteFrame::create("Blue_Front1.png", Rect(0, 0, 65, 81)));
...
    Animation* animation = Animation::createWithSpriteFrames(animFrames, 0.1f);
```

Зверніть увагу на створення вказівника **animation** тут вказується весь вектор спрайтів, а також в якості аргументів вказується час затримки кожного спрайту.

Ще один приклад створення анімації з спрайт-фрейму та використання класу вектор (Vector):

```
Vector HelloWorld::getAnimation(const char *format, int count)
{
    auto spritecache = SpriteFrameCache::getInstance();
    Vector animFrames;
    char str[100];
    for (int i = 1; i <= count; i++)
    {
        sprintf(str, format, i);
```

```
                    animFrames.pushBack(spritecache-
>getSpriteFrameByName(str));
                }
                return animFrames;
        }
```

Вищеописаний метод витягує кадри анімації зі спрайт-списку, що йдуть за заданим форматом. Його можна викликати наступним чином:

```
Vector frames = getAnimation("run/run_%002d.png ", 8);
```

Для реалізації анімації використовуючи вищеописаний метод **getAnimation** потрібно дописати наступне:

```
auto frames = getAnimation("run/run_%002d.png ", 8);
auto sprite = Sprite::createWithSpriteFrame(frames.front());
background->addChild(sprite);
sprite->setPosition(100, 620);

auto animation = Animation::createWithSpriteFrames(frames, 1.0f / 8);
sprite->runAction(RepeatForever::create(Animate::create(animation)));
```

Прискорення та затримка руху (Easing)

Основою кожної гри є анімація, рух. Чим рухи природніші тим гра професійніша, правдивіша. Основою кожного руху є правильні рухові елементи, так звана фізика гри.

Для реалізації фізичних властивостей існують свої методи та функції. Проте можна використати деякі стандартні анімації, які можуть замінити фізичні властивості елементів.

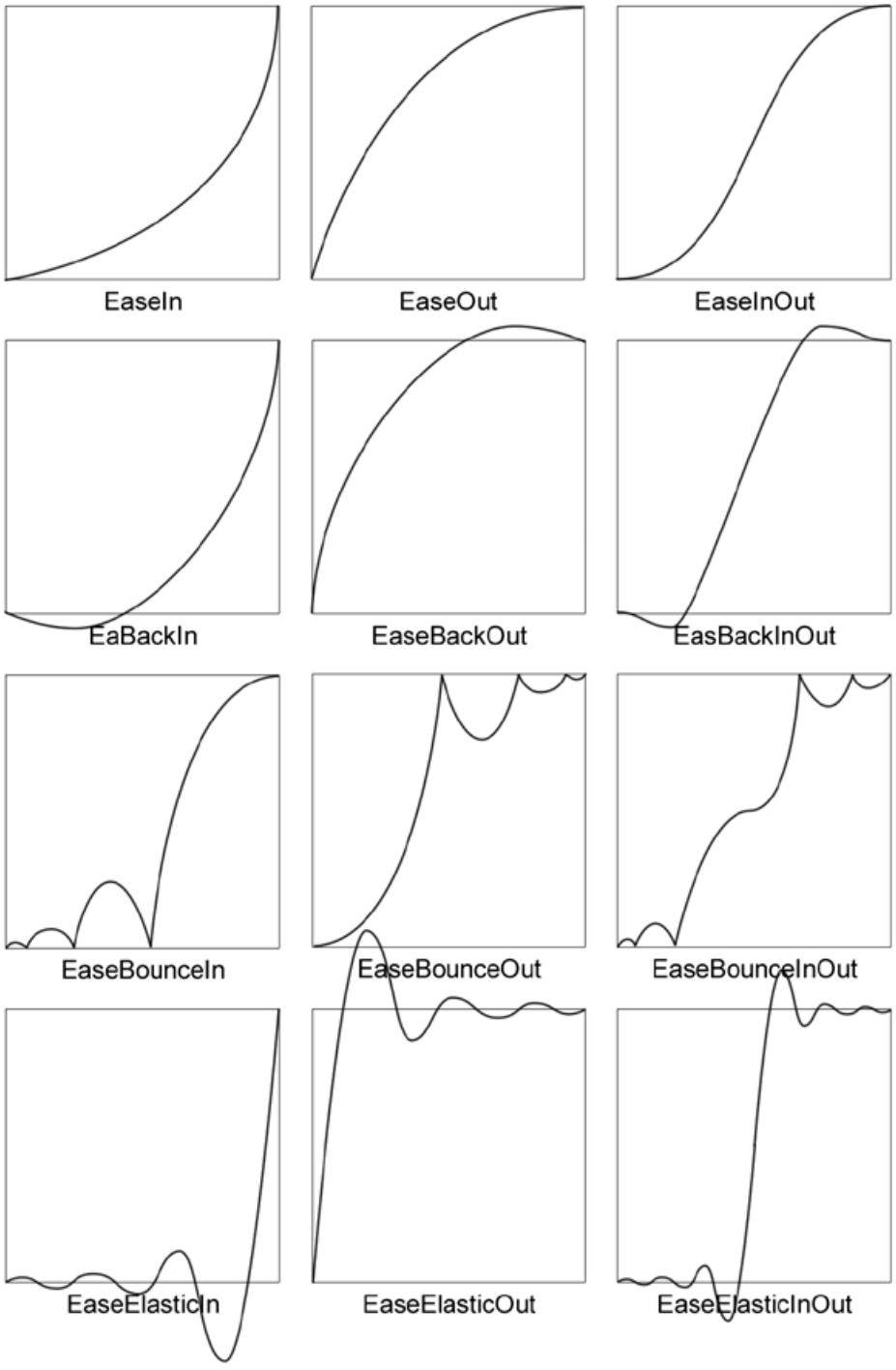
Скажімо ефект відбивання м'ячика від твердої поверхні, які поволі затухають. Використання такої анімації надасть природності грі.

Описані можливості анімації можуть використовуватись із прискоренням чи затримкою: EaseIn і EaseOut. Може починатись з прискоренням, а закінчуватись із затримкою EaseInOut.

Таблиця 2 Класи прискорення та затримки

EaseIn	Рух з прискоренням
EaseOut	Рух з затуханням
EaseInOut	Мочаток руху з прискоренням, завершення з затуханням
EaseExponentialIn	Це схожа дія на EaseIn, але прискорення відбувається експоненціально. Також використовується з Out та InOut як EaseIn.

EaseSineIn	Це схоже на EaseIn, але має на меті синусоїдальну зміну швидкості. Також використовується з Out та InOut аналогічно як з EaseIn.
EaseElasticIn	Рух поступово все помаліше та помаліше. Також використовується з Out та InOut аналогічно як з EaseIn.
EaseBounceIn	Рух після підстрибувань. Також використовується з Out та InOut аналогічно як з EaseIn.
EaseBackIn	Рухається після руху у зворотному напрямку. Також використовується з Out та InOut аналогічно як з EaseIn.



мал. 4

Тривалість анімації, не залежить від типу анімації. EaseIn, EaseOut і EaseInOut мають два параметри: перший параметр - це дія ослаблення, другий параметр - швидкість дії.

На малюнках 4 подано найтиповіші класи, з характером руху:

Приклади.

Використання EaseInOut

```
auto sprite = Sprite::create("res/run_01.png");
sprite->setPosition(Vec2(200, 200));
this->addChild(sprite);
auto move = MoveTo::create(3.0f, Vec2(500, 200));
auto ease = EaseInOut::create(move, 2.0f);
sprite->runAction(ease);
```

Використання **EaseBounceOut**

```
        auto sprite = Sprite::create("res/run_01.png");
sprite->setPosition(Vec2(size.width / 2, size.height));
sprite->setAnchorPoint(Vec2(0.5f, 0.0f));
this->addChild(sprite);
auto drop = MoveTo::create(3.0f, Vec2(size.width / 2, 0));
auto ease = EaseBounceOut::create(drop);
sprite->runAction(ease);
```

Використання ефекту затухаючих чи наростаючих рухів можна використовувати в складних послідовностях:

```
auto move = MoveBy::create(2, Vec2(200, 200));
auto move_back = move->reverse();

auto move_ease_in = EaseQuadraticActionInOut::create(move->clone());

auto delay = DelayTime::create(0.25f);

auto seq1 = Sequence::create(move_ease_in, delay, move_back,
        delay->clone(), nullptr);

mySprite->runAction(RepeatForever::create(seq1));
```

auto move_back = move->reverse(); - формується анімація зворотнього руху до **auto move = MoveBy::create(2, Vec2(200, 200));**

Послідовності

Послідовності - це серія рухів об'єктів, які виконуються послідовно. Елементами послідовності можуть бути, в довільній кількості, рухи об'єктів, функції інші послідовність.

Якщо б ви мали виконувати кілька дій одна за одною, вони все почали б одночасно:

```
mySprite->setPosition(Vec2(250, 250));
auto jump = JumpBy::create(0.5, Vec2(0, 0), 100, 1);
auto rotate = RotateBy::create(2.0f, 90);\
mySprite->runAction(jump);
mySprite->runAction(rotate);
```

Це, за певних обставин, може дати бажаний ефект. Але в деяких випадках виникає потреба у виконанні дії, яка повинна виконуватися лише одна, і тільки після її завершення повинна запускатись наступна дія, і так далі.

Це можна досягти, використовуючи Sequencing дія, яка по суті містить будь-яку комбінацію інших дій.

Дія «послідовність» виконується наступним чином:

• Послідовність: ця дія виконує кожну визначену дію, одну за одною, після повного закінчення попередньої дії.

Приклад:

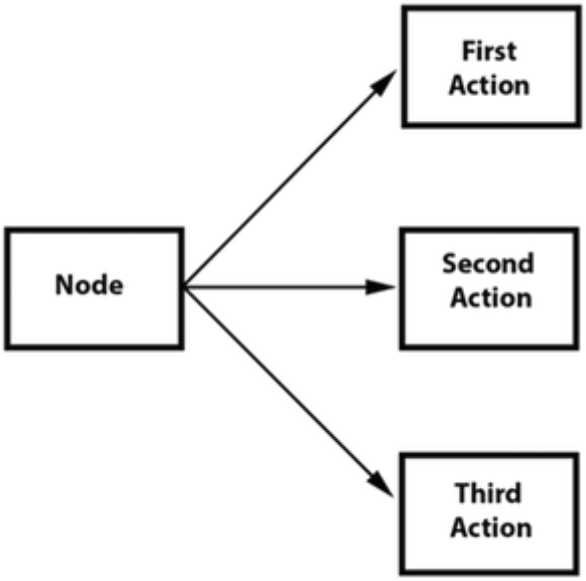
```
auto jump = JumpBy::create(0.5, Vec2(0, 0), 100, 1);

auto rotate = RotateBy::create(2.0f, 90);
auto delay1 = DelayTime::create(2.0f);

auto seq = Sequence::create(jump, delay1, rotate, delay1, nullptr);
mySprite->runAction(RepeatForever::create(seq));
        scene->addChild(mySprite);
```

Схема виконання послідовності, може виглядати наступним чином:

мал. 5

Spawn - послідовності

Spawn за своєю дією дуже схожий на послідовність, за винятком того, що всі дії будуть виконуватися одночасно. Ви можете мати будь-яку кількість об'єктів-дій та навіть інших об'єктів Spawn!



мал. 6

Spawn виробляє такий же результат, як виконання декількох послідовних операторів runAction(). Проте перевага spawn полягає в тому, що ви можете помістити його в послідовність, щоб допомогти досягти певних ефектів, які ви не можете інакше. Поєднання Spawn та послідовності є дуже потужною функцією.

Наприклад:

Створюємо два об'єкти дії:

```
auto mySprite = Sprite::create("1.png");
auto moveBy = MoveBy::create(10, Vec2(400, 100));
auto fadeTo = FadeTo::create(8.0f, 20.0f);
```

Далі створюється об'єкт класу Spawn, а далі цей об'єкт передається в функцію `runAction()`:

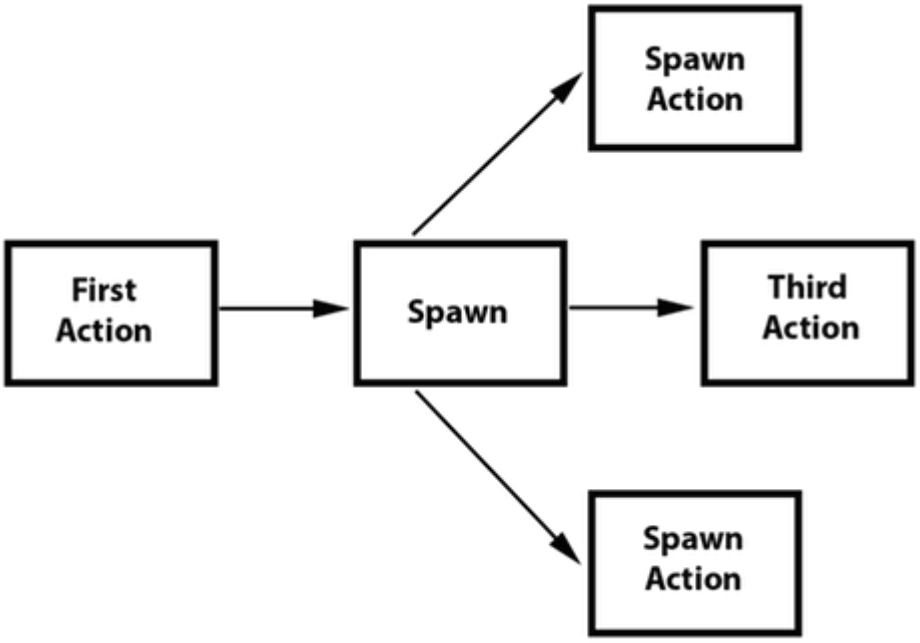
```
auto mySpawn = Spawn::createWithTwoActions(moveBy, fadeTo);
mySprite->runAction(mySpawn);
```

аналогічну дію можна було зробити використавши два рази функцію **runAction()**:

```
mySprite->runAction(moveBy);
mySprite->runAction(fadeTo);
```

Вищенаведений код забезпечить одночасність виконання дій.

Окрім того Spawn можна використовувати як елемент послідовності (Sequence). Таку реалізацію можна проілюструвати наступною схемою:



мал. 7

```
// створення кількох дій
    auto moveBy = MoveBy::create(10, Vec2(400, 100));
    auto fadeTo = FadeTo::create(2.0f, 120.0f);
    auto scaleBy = ScaleBy::create(2.0f, 3.0f);

    // створення Spawn послідовності
    auto mySpawn = Spawn::createWithTwoActions(scaleBy, fadeTo);

    // компанування все в єдину послідовність
    auto seq = Sequence::create(moveBy, mySpawn, moveBy, nullptr);

    // запуск
    mySprite->runAction(seq);
```

Clone

Clone – метод, який дозволяє використовувати одну і туж дію повторно кілька раз, застосовуючи її до різних об'єктів. Якщо є дія, її можете застосувати до об'єктів декількох вузлів за допомогою Clone(). Об'єкти дії мають внутрішній стан. Коли вони запускаються, вони фактично змінюють властивості об'єктів вузла. Без використання Clone() втрачається унікальність дії, і повторне використання до об'єкту може привезти до непередбачуваних дій. Це призведе до несподіваних результатів, оскільки ви не можете точно дізнатися, які властивості дії наразі встановлено.

Давайте проведемо приклад, кажуть, що у вас є спрайт, і він має позицію (0,0). Якщо ви запускаєте дії:

```
MoveBy::create(10, Vec2(400, 100));
```

Це перемістить Sprite від (0,0) до (400, 100) протягом 10 секунд. Sprite тепер має нову позицію (400, 100) і, що важливіше, Action має цю позицію у його

внутрішньому стані. Тепер скажіть, у вас є інший Sprite з позицією (200, 200). Якщо б ви мали застосовувати цю саму дію:

```
MoveBy::create(10, Vec2(400, 100));
```

до цього спрайту, то він опинився б на позиції (800, 200), а не там, де б логічно мало бути (600, 300).

Це відбувається тому, що Дія вже мала внутрішній стан, щоб почати виконання функції MoveBy. Клонування дії запобігає цьому. Це гарантує, що ви отримаєте унікальну версію Action, застосовану до вашого вузла.

Ось як має виглядати правильний код:

```
auto heroSprite = Sprite::create("herosprite.png");
auto enemySprite = Sprite::create("enemysprite.png");

auto moveBy = MoveBy::create(10, Vec2(400, 100));

heroSprite->runAction(moveBy);
enemySprite->runAction(moveBy->clone());
```

Реверс

Якщо в програмі виконуєте ряд дій, то до них можна викликати метод reverse() , щоб запустити дію, у зворотному порядку. В іншому випадку відомий як, назад. Однак, це не просто запуск програми в зворотному порядку. Виклик reverse() насправді маніпулює властивостями оригінальної послідовності(Sequence) або Spawn-послідовності виконуючи їх в зворотньому порядку.

Використання методу reverse() є досить простим, наприклад до Spawn-послідовності з попереднього прикладу:

```
mySprite->runAction(mySpawn->reverse());
```

Таким чином використання зворотніх дій є досить простим. Це легко переконатись запустивши наступний код:

```
auto moveBy = MoveBy::create(2.0f, Vec2(500, 0));
auto scaleBy = ScaleBy::create(2.0f, 2.0f);
auto delay = DelayTime::create(2.0f);

// створення послідовностей
auto delaySequence = Sequence::create(delay, delay->clone(), delay->clone(),
delay->clone(), nullptr);

auto sequence = Sequence::create(moveBy, delay, scaleBy, delaySequence,
nullptr);

auto sequence1 = Sequence::create(sequence, sequence->reverse(), nullptr);

// запуск
```

```
mySprite->runAction(sequence1);
scene->addChild(mySprite);
```

Список літературних джерел по темі:

1. **Siddharth Shekar** Learning Cocos2d-x Game Development // 2014 Packt Publishing, p.266, ISBN 978-1-78398-826-6
2. Raydelto Hernandez Building Android Games with Cocos2d-x // 2015 Packt Publishing, p. 160, ISBN 978-1-78528-383-3
3. Roger Engelbert Cocos2d-x by Example Beginner's Guide Second Edition // 2015 Packt Publishing, p. 270, ISBN 978-1-78528-885-2
4. Akihiro Matsuura Cocos2d-x Cookbook // 2015 Packt Publishing, p. 255, ISBN 978-1-78328-475-7
5. Karan Sequeira Cocos2d-x Game Development Blueprints // 2015 Packt Publishing, p. 392, ISBN 978-1-78398-526-5
6. Frahaan Hussain, Arutosh Gurung, Gareth Jones Cocos2d-x Game Development Essentials // 2014 Packt Publishing, p. 392, ISBN 978-1-78398-786-3